Progressive Join Algorithms Considering User Preference

Mengsu Ding, **Shimin Chen***

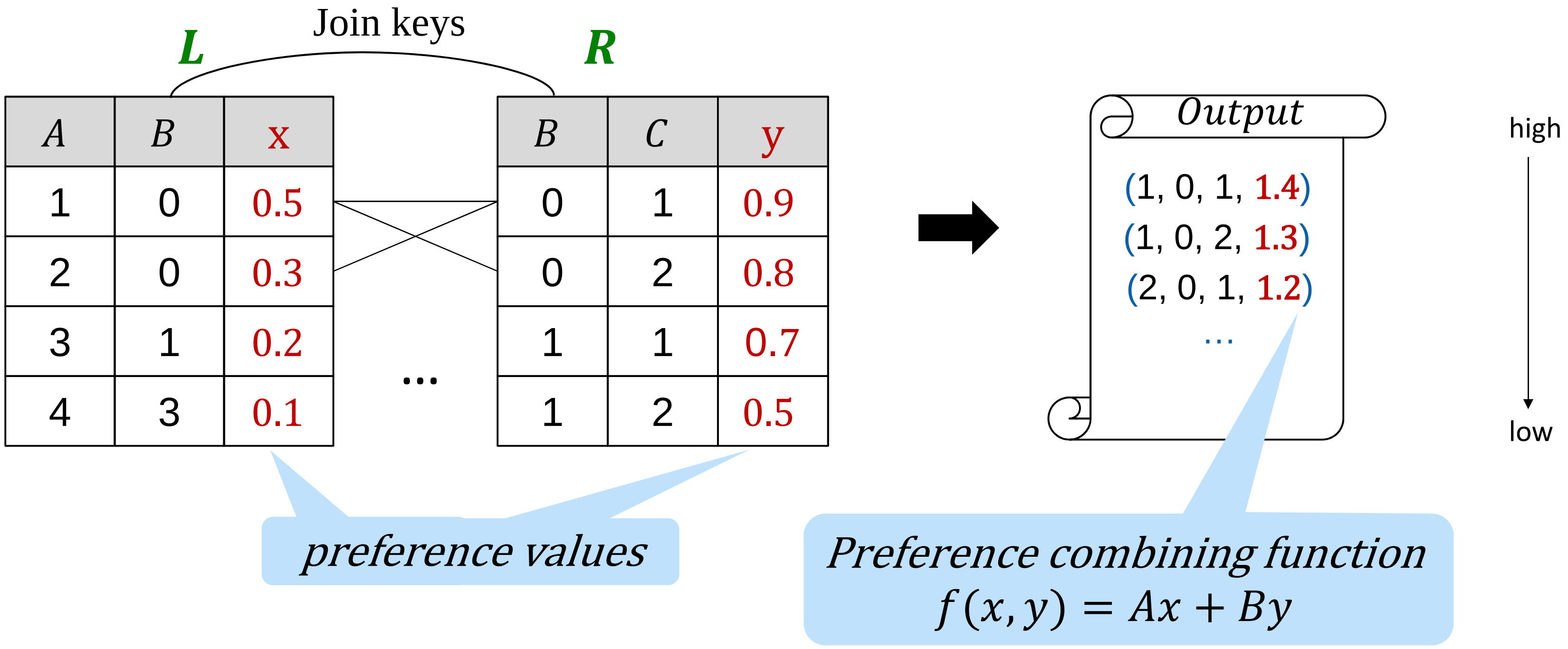
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China

Nantia Makrynioti, Stefan Manegold

CWI (Centrum Wiskunde & Informatica)
Amsterdam, the Netherlands

Preference-aware Progressive Join

- Progressive query processing for exploratory data analysis
- Return join results ordered according to preference



Join keys

L

A	*B*	x
1	0	0.5
2	0	0.3
3	1	0.2
4	3	0.1

…

R

B	*C*	y
0	1	0.9
0	2	0.8
1	1	0.7
1	2	0.5

Output

(1, 0, 1, 1.4)
(1, 0, 2, 1.3)
(2, 0, 1, 1.2)
…

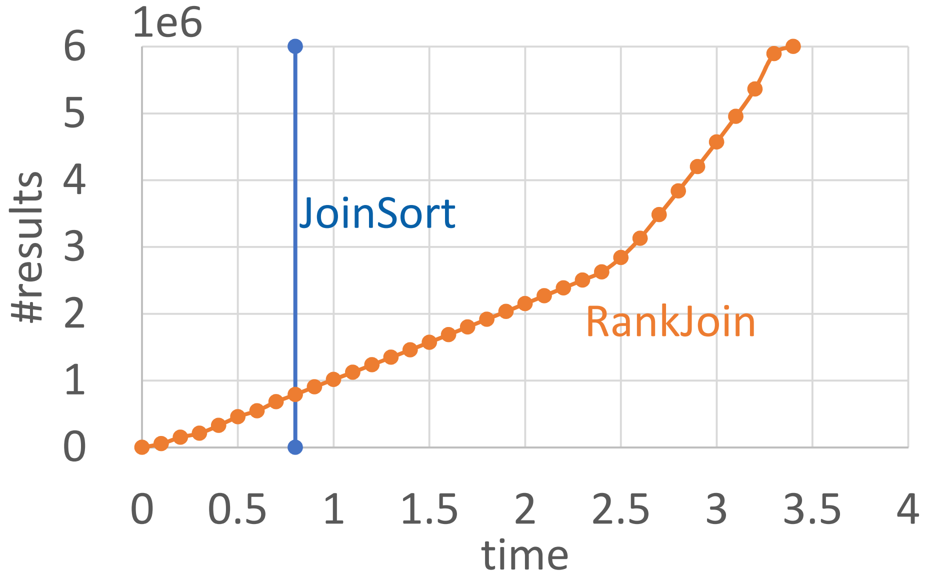
preference values

Preference combining function

$f(x, y) = Ax + By$

- Goal: Fast early results & Fast full results

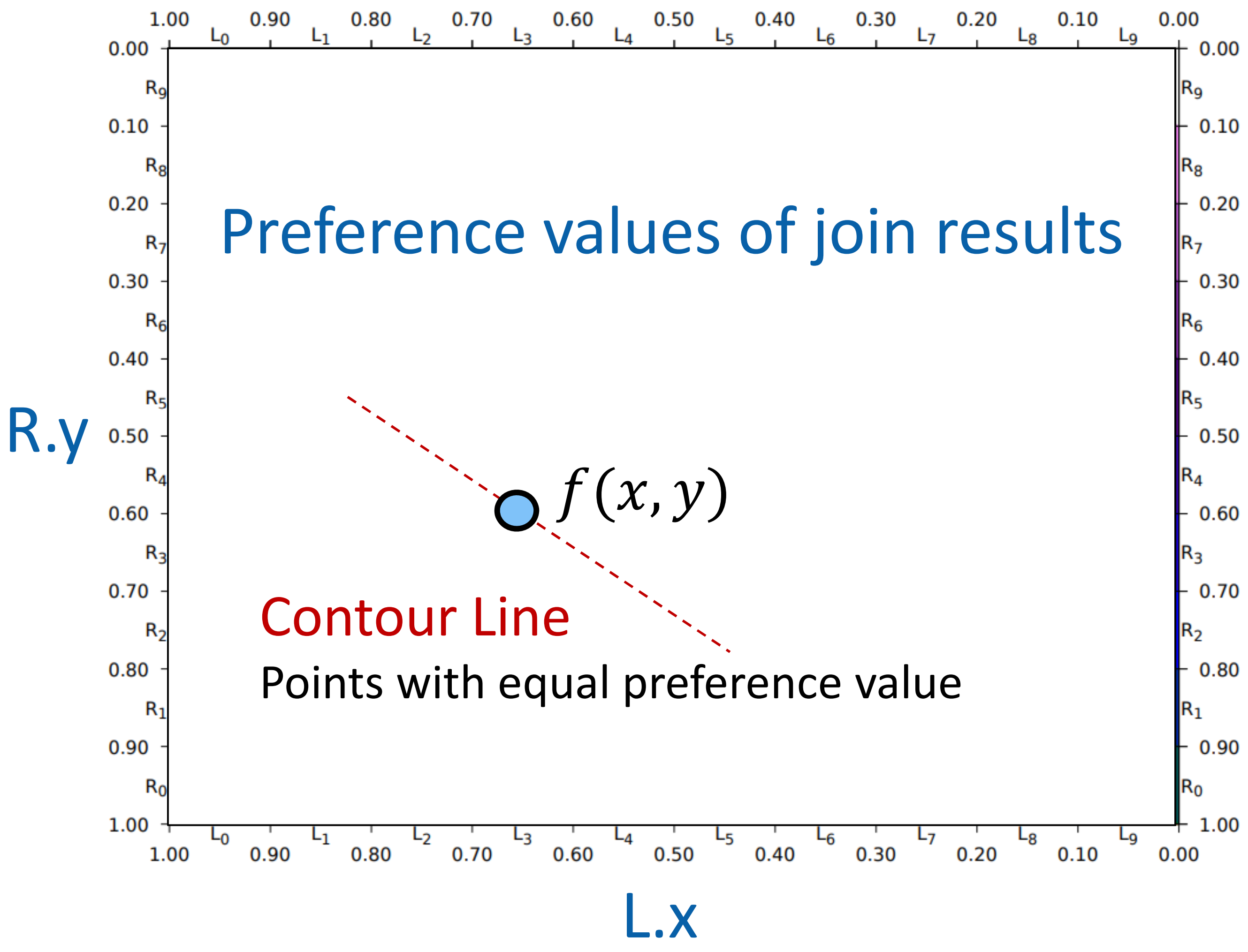
Problems of Existing Solutions



- Blocking approach: Join + Sort
  - ❑ **No early results**
- Extending top-k join algorithms:
  - ❑ e.g., RankJoin [VLDB'03] symmetric hash join + priority queue
  - ❑ **Slow full results**
  - ❑ **Significant sorting overhead**

We want to reduce or eliminate sorting overhead

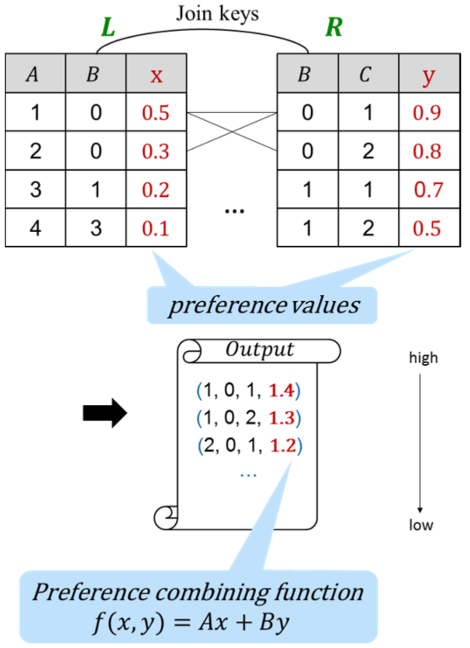
Our Idea: Exploiting Contour Lines



Preference values of join results

Contour Line

Points with equal preference value



A	B	x
1	0	0.5
2	0	0.3
3	1	0.2
4	3	0.1

B	C	y
0	1	0.9
0	2	0.8
1	1	0.7
1	2	0.5

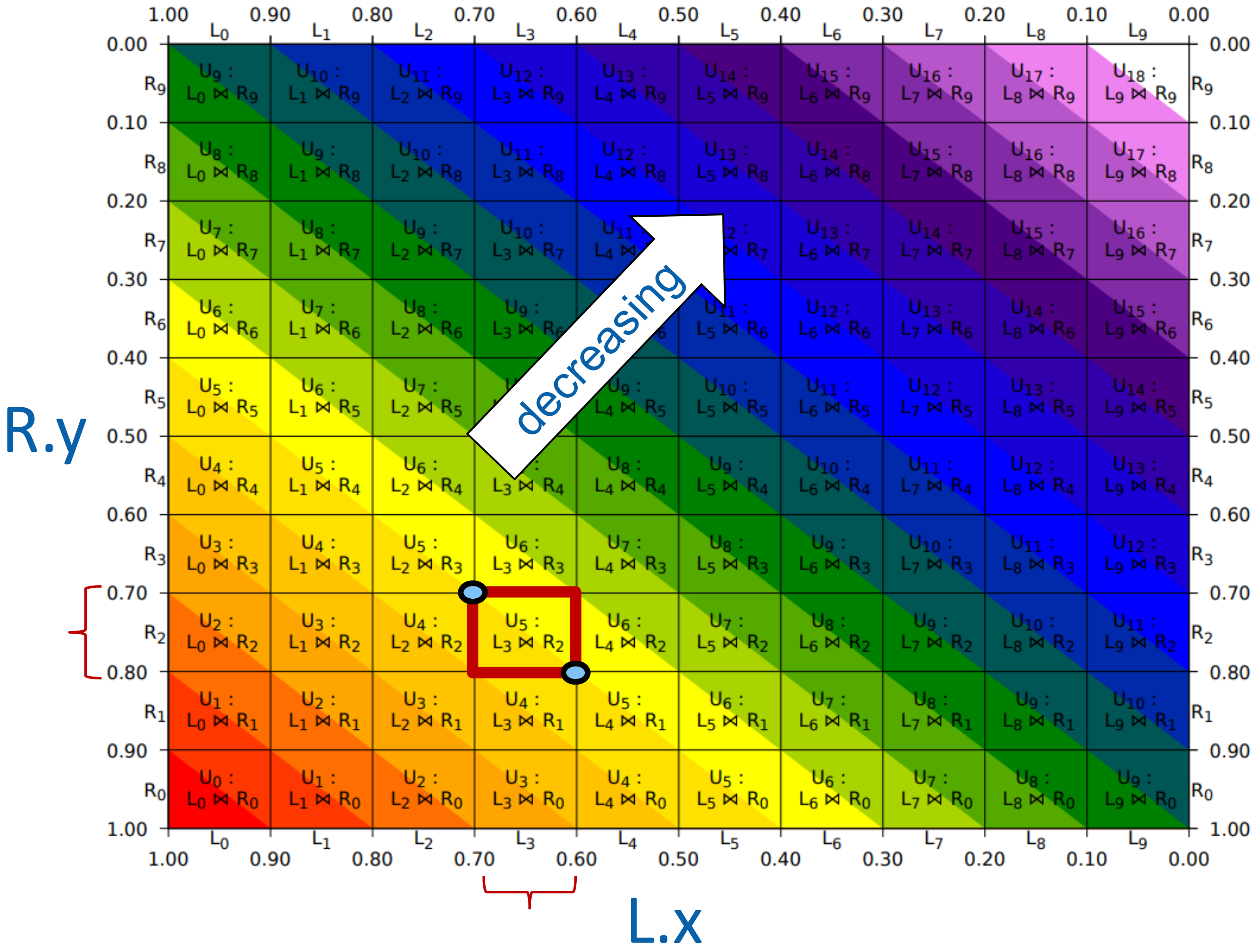
preference values

Output

(1, 0, 1, 1.4)
(1, 0, 2, 1.3)
(2, 0, 1, 1.2)
…

Preference combining function
$f(x,y) = Ax + By$

Our Idea: Exploiting Contour Lines



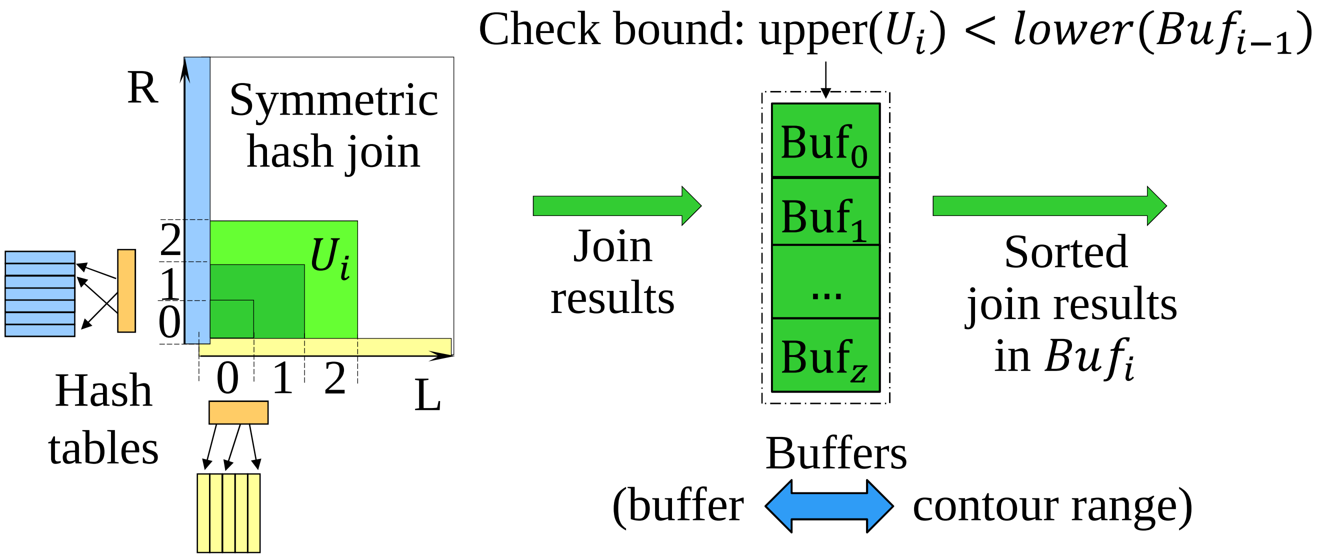
	L_0	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9
R_9	U_9 : $L_0 \bowtie R_9$	U_{10} : $L_1 \bowtie R_9$	U_{11} : $L_2 \bowtie R_9$	U_{12} : $L_3 \bowtie R_9$	U_{13} : $L_4 \bowtie R_9$	U_{14} : $L_5 \bowtie R_9$	U_{15} : $L_6 \bowtie R_9$	U_{16} : $L_7 \bowtie R_9$	U_{17} : $L_8 \bowtie R_9$	U_{18} : $L_9 \bowtie R_9$
R_8	U_8 : $L_0 \bowtie R_8$	U_9 : $L_1 \bowtie R_8$	U_{10} : $L_2 \bowtie R_8$	U_{11} : $L_3 \bowtie R_8$	U_{12} : $L_4 \bowtie R_8$	U_{13} : $L_5 \bowtie R_8$	U_{14} : $L_6 \bowtie R_8$	U_{15} : $L_7 \bowtie R_8$	U_{16} : $L_8 \bowtie R_8$	U_{17} : $L_9 \bowtie R_8$
R_7	U_7 : $L_0 \bowtie R_7$	U_8 : $L_1 \bowtie R_7$	U_9 : $L_2 \bowtie R_7$	U_{10} : $L_3 \bowtie R_7$	U_{11} : $L_4 \bowtie$	2 : $\bowtie R_7$	U_{13} : $L_6 \bowtie R_7$	U_{14} : $L_7 \bowtie R_7$	U_{15} : $L_8 \bowtie R_7$	U_{16} : $L_9 \bowtie R_7$
R_6	U_6 : $L_0 \bowtie R_6$	U_7 : $L_1 \bowtie R_6$	U_8 : $L_2 \bowtie R_6$	U_9 : $L_3 \bowtie R_6$	6	U_{11} : $L_5 \bowtie R_6$	U_{12} : $L_6 \bowtie R_6$	U_{13} : $L_7 \bowtie R_6$	U_{14} : $L_8 \bowtie R_6$	U_{15} : $L_9 \bowtie R_6$
R_5	U_5 : $L_0 \bowtie R_5$	U_6 : $L_1 \bowtie R_5$	U_7 : $L_2 \bowtie R_5$	U	U_9 : $L_4 \bowtie R_5$	U_{10} : $L_5 \bowtie R_5$	U_{11} : $L_6 \bowtie R_5$	U_{12} : $L_7 \bowtie R_5$	U_{13} : $L_8 \bowtie R_5$	U_{14} : $L_9 \bowtie R_5$
R_4	U_4 : $L_0 \bowtie R_4$	U_5 : $L_1 \bowtie R_4$	U_6 : $L_2 \bowtie R_4$	: $L_3 \bowtie R_4$	U_8 : $L_4 \bowtie R_4$	U_9 : $L_5 \bowtie R_4$	U_{10} : $L_6 \bowtie R_4$	U_{11} : $L_7 \bowtie R_4$	U_{12} : $L_8 \bowtie R_4$	U_{13} : $L_9 \bowtie R_4$
R_3	U_3 : $L_0 \bowtie R_3$	U_4 : $L_1 \bowtie R_3$	U_5 : $L_2 \bowtie R_3$	U_6 : $L_3 \bowtie R_3$	U_7 : $L_4 \bowtie R_3$	U_8 : $L_5 \bowtie R_3$	U_9 : $L_6 \bowtie R_3$	U_{10} : $L_7 \bowtie R_3$	U_{11} : $L_8 \bowtie R_3$	U_{12} : $L_9 \bowtie R_3$
R_2	U_2 : $L_0 \bowtie R_2$	U_3 : $L_1 \bowtie R_2$	U_4 : $L_2 \bowtie R_2$	U_5 : $L_3 \bowtie R_2$	U_6 : $L_4 \bowtie R_2$	U_7 : $L_5 \bowtie R_2$	U_8 : $L_6 \bowtie R_2$	U_9 : $L_7 \bowtie R_2$	U_{10} : $L_8 \bowtie R_2$	U_{11} : $L_9 \bowtie R_2$
R_1	U_1 : $L_0 \bowtie R_1$	U_2 : $L_1 \bowtie R_1$	U_3 : $L_2 \bowtie R_1$	U_4 : $L_3 \bowtie R_1$	U_5 : $L_4 \bowtie R_1$	U_6 : $L_5 \bowtie R_1$	U_7 : $L_6 \bowtie R_1$	U_8 : $L_7 \bowtie R_1$	U_9 : $L_8 \bowtie R_1$	U_{10} : $L_9 \bowtie R_1$
R_0	U_0 : $L_0 \bowtie R_0$	U_1 : $L_1 \bowtie R_0$	U_2 : $L_2 \bowtie R_0$	U_3 : $L_3 \bowtie R_0$	U_4 : $L_4 \bowtie R_0$	U_5 : $L_5 \bowtie R_0$	U_6 : $L_6 \bowtie R_0$	U_7 : $L_7 \bowtie R_0$	U_8 : $L_8 \bowtie R_0$	U_9 : $L_9 \bowtie R_0$

f(L.x,R.y) ≥ 0.0	⬜
f(L.x,R.y) ≥ 0.1	🟪
f(L.x,R.y) ≥ 0.2	🟪
f(L.x,R.y) ≥ 0.3	🟪
f(L.x,R.y) ≥ 0.4	🟪
f(L.x,R.y) ≥ 0.5	🟦
f(L.x,R.y) ≥ 0.6	🟦
f(L.x,R.y) ≥ 0.7	🟦
f(L.x,R.y) ≥ 0.8	🟦
f(L.x,R.y) ≥ 0.9	🟩
f(L.x,R.y) ≥ 1.0	🟩
f(L.x,R.y) ≥ 1.1	🟩
f(L.x,R.y) ≥ 1.2	🟩
f(L.x,R.y) ≥ 1.3	🟨
f(L.x,R.y) ≥ 1.4	🟨
f(L.x,R.y) ≥ 1.5	🟧
f(L.x,R.y) ≥ 1.6	🟧
f(L.x,R.y) ≥ 1.7	🟧
f(L.x,R.y) ≥ 1.8	🟥
f(L.x,R.y) ≥ 1.9	🟥

$$f(x, y) = Ax + By$$

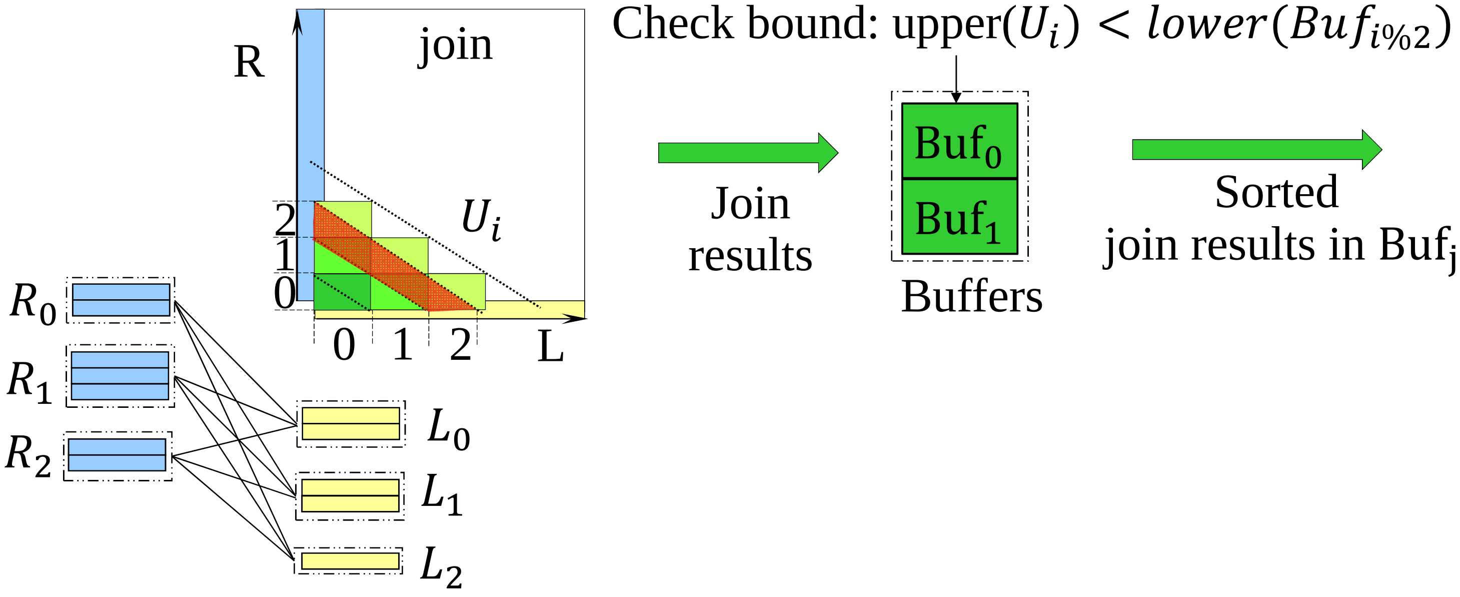
A=B=1

Inputs Follow Contour Lines (Algorithm1: CJpI)



- Avoid sorting across buffers
- Only need to sort within each buffer

Both Inputs and Outputs Follow Contour Lines (Algorithm2: CJpB)



Check bound: upper(U_i) $< lower(Buf_{i\%2})$

Join results

Buffers

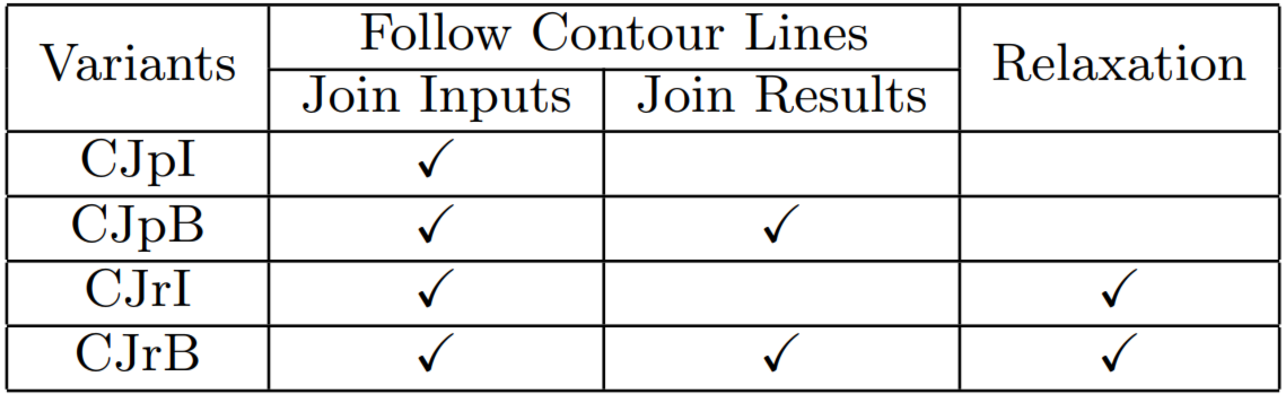
Sorted join results in Buf_j

- Reduce intermediate result size
- But join cost may be higher

Relaxation to Remove Sorting

- Relaxation of Order: **no intra-buffer sorting**
  - ❑ Relaxation: tuples t_i and t_j are regarded as in order if $\left|t_i.pval - t_j.pval\right| \leq \epsilon$
  - ❑ Judiciously select input intervals

4 Variants of Contour Joins



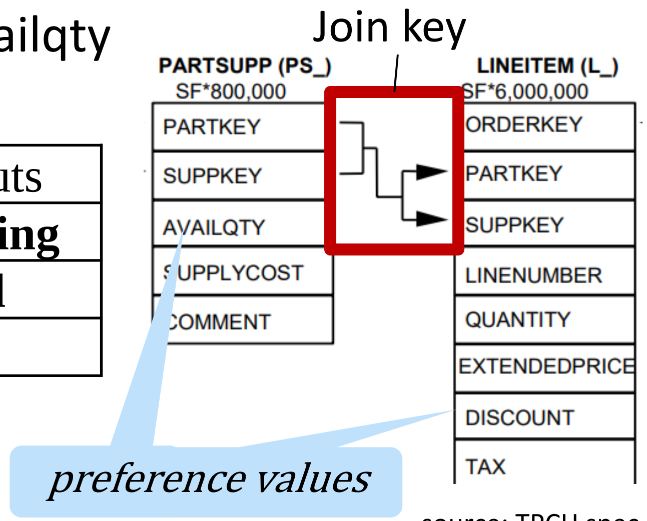
Variants	Follow Contour Lines		Relaxation
	Join Inputs	Join Results	
CJpI	✓		
CJpB	✓	✓	
CJrI	✓		✓
CJrB	✓	✓	✓

p: precise, r: relaxed; I: Inputs, B: Both inputs & outputs

Experiments

- Data Set: Based on TPC-H Lineitem and Partsupp
  - ❑ Preference values based on: l_discount, ps_availqty
  - ❑ 3 datasets (can fit into main memory)

Scale Factor	Datasets	#Inputs	#Outputs
m_1	*datasets#1*	**fixed**	**increasing**
m_2	*datasets#2*	**increasing**	**fixed**
m_3	*datasets#3*	**increasing**	



Join key

PARTSUPP (PS_)
SF*800,000
PARTKEY
SUPPKEY
AVAILQTY
SUPPLYCOST
COMMENT

LINEITEM (L_)
SF*6,000,000
ORDERKEY
PARTKEY
SUPPKEY
LINENUMBER
QUANTITY
EXTENDEDPRICE
DISCOUNT
TAX

preference values

source: TPCH spec

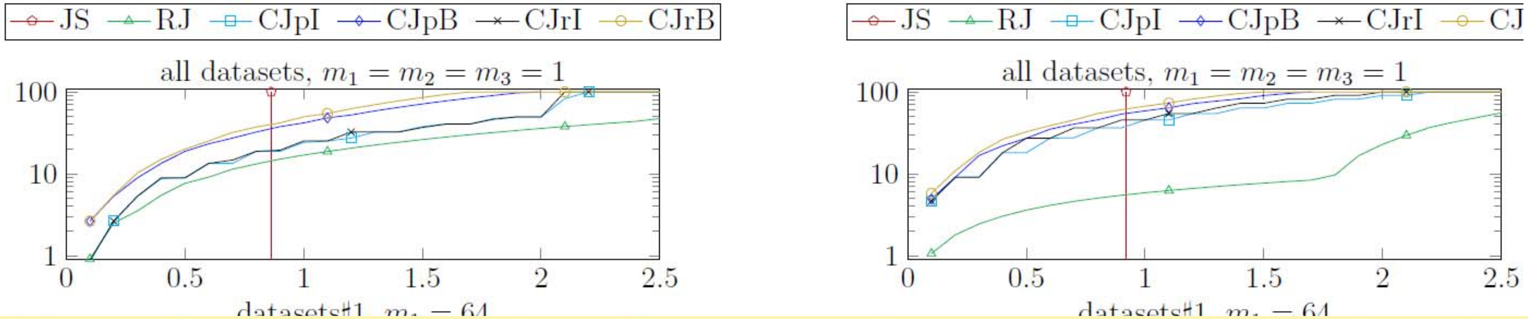
- Query

$f(x, y) = Ax + By$

select L.key, $f(L.pval, PS.pval)$ as score
from Lineitem as L, Partsupp as PS
where L.key = PS.key
order by score ASC
progressively

- Machine
  - ❑ Intel Core i7-4770 CPU @3.40GHz (8MB cache) and 32GB memory
  - ❑ 64-bit Ubuntu 16.04 LTS with 4.15.0-62-generic Linux kernel

Overall Results



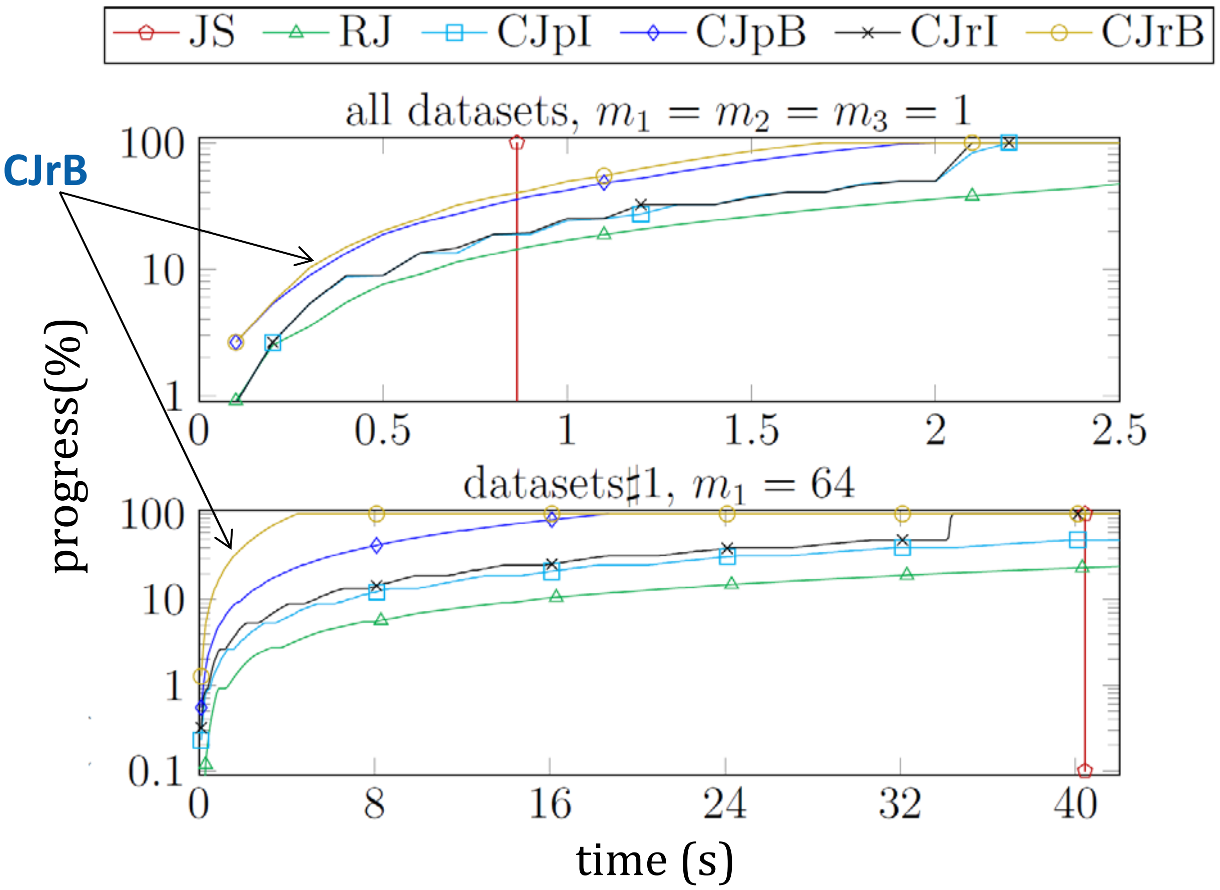
- **Compared to RJ (RankJoin)**
  - ❑ 1% early results
    - – best **precise** contour join: up to 7x improvements
    - – best **relaxed** contour join: up to 14x improvements
  - ❑ Full results
    - – best **precise** contour join: up to 10.6x improvements
    - – best **relaxed** contour join: up to 39.4x improvements
- **Comparable or better perfomance to JS (blocking JoinSort)**

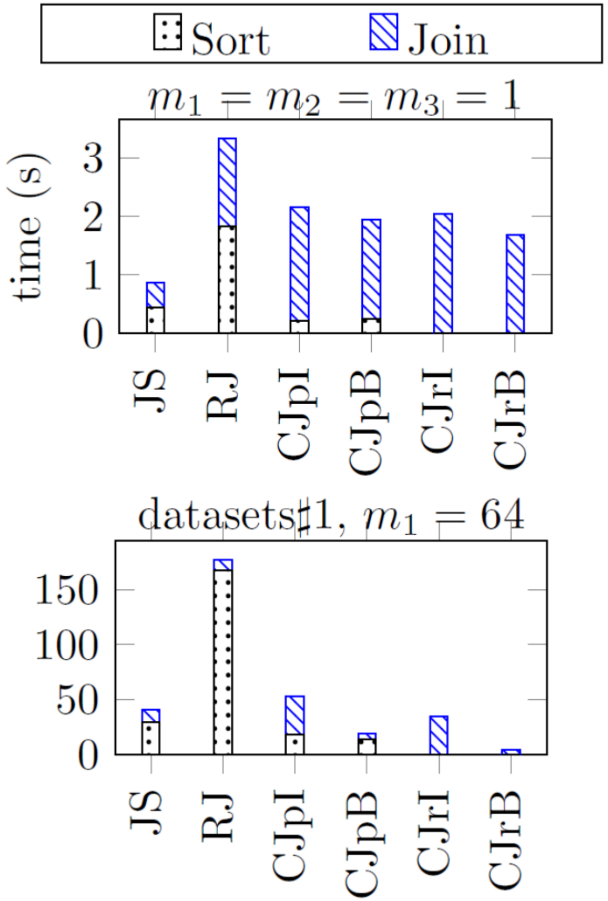
10
1
0
8
16
24
32
40
48
56
time(s)
10
1
0
8
16
24
32
40
48
56
time (s)

(a) $f(x, y) = x + y$, $p_L = 200$, $p_R = 200$

(b) $f(x, y) = 10x + y$, $p_L = 2000$, $p_R = 200$

Fix Input Size, Increase Join Result Size





- As join result size increases, the fraction of sorting increases
- RankJoin becomes very poor
- CJrB is the best performing algorithm

Conclusion

- Preference-aware joins in progressive query processing
- Idea: exploit contour lines in the join result space
- ContourJoin: a promising solution
  - ❑ Faster early and full results generation (vs. RankJoin)
  - ❑ Good total result generation performance (vs. JoinSort)

❑ More in the paper

- Algorithms and proofs
- Extensive experimental results
- Discussion on preference combining functions, unsorted inputs, multi-way joins